Applying AI Techniques to Program Optimization For Parallel Computers

By

Ko-Yang Wang
Department of Computer Science
Purdue University
West Lafayette, Indiana

and

Dennis Gannon
Department of Computer Science
Indiana University
Bloomington, Indiana

TECHNICAL REPORT NO. 227

Applying AI Techniques to Program Optimization For Parallel Computers

by

Ko-Yang Wang and Dennis Gannon

September, 1987

This report is based on work supported in part by the Air Force Office of Scientific Research under Grant No. AFOSR-86-0147.

12.1 Introduction

12.1.1 The Trend Toward Parallelism

Perhaps the most important trend in supercomputer design is the reliance on parallelism to achieve performance improvements over our fastest sequential processors. During the three-year period from 1984 to 1987, the number of commercially available general purpose parallel processing systems jumped from a couple to over a dozen. The number of ways in which different architectures exploit parallelism is almost as large as the number of different companies. This is a healthy situation for computer architecture. Many good ideas are emerging. Unfortunately, each different machine presents a different architectural model to the programmer. A program that has been optimized for one system may not be well suited to another. At first glance, the differences may appear to be due to the fact that each machine supports a different set of extensions to FORTRAN, or even a different base programming language. But a deeper analysis shows that the architectural difference between machines plays a fundamental role in the organization of the computation. Surface level syntactic changes are not enough to port a program optimized for a Cray XMP to good code for a MIMD hypercube design. While this is an extreme case, it illustrates the problems faced by the small, but growing, cadre of programmers who have taken up the task of putting these machines to productive use.

Because of these problems, it has become clear that the greatest need in supercomputer development is a new generation of software tools that can help in the task of optimizing code for new architectures.

In this chapter, we describe a project under development at Purdue University and Indiana University, which is an experiment in integrating expert systems technology with the advanced compiler optimization research conducted over the last ten years by Kuck, Wolfe, and their associates in Urbana Illinois [3, 13, 14, 15, 20, 21, 23, 29], Kennedy and his students at Rice [11, 1], and Allan, Cytron, and Burke at Yorktown Heights [6, 4]. There are three key ideas that are guiding our work:

- Interactive program restructuring tools are essential in helping users move programs to new machines.
- Expert knowledge about how to choose a sequence of restructuring transformations that optimize performance can be organized as an "advice giving" system. Furthermore, performance models of the target architecture can be incorporated into a rule based system to guide the transformation process.

- New architectural models and expert programming heuristics for new target machines must be easily incorporated into such a system in a uniform manner.

Of course, interactive tools already exist. For example, FORGE from Pacific-Sierra Research provides an excellent user interface. PTOOL from Rice University [2] has an elegant way to help users identify data dependence in programs. And all automatic program restructurers, such as VAST, KAP and Paraphrase, employ powerful heuristics to retarget user code. The goal of this research is to show that an expert systems approach is a more flexible and extensible model than the conventional parallel compilers for designing a tool that can be rapidly adapted to new target machines and new heuristics for parallel program optimization.

12.1.2 Automatic Program Parallelism Optimization

The program parallelism optimization problem is the following: given a program and a target parallel machine, how can a parallel program that is both functionally equivalent to the original program and optimal for the target machine be generated?

The basic algorithm for program parallelism optimization can be outlined as the following:

BasicProgram

Input: a sequential or parallel program and the description of the target machine.

Output: a parallel program that is optimal for the target machine.

```
Begin
    repeat
        pick the "best" transformation from the set of
                all applicable transformations;
        apply the selected transformation to the program;
    until the resulting program is optimal for the target machine
End;
```

This algorithm is superficial in the sense that it does not specify how to determine either which transformation is the best or when the program is optimal. However, this is the algorithm that most parallel computer users use when they hand-optimize their programs. Picking the "best" transformation requires expert intelligence.

Our goal is to design an intelligent system that can perform the program parallelism optimization process for different classes of target machines automatically. Several fundamental issues must be addressed before such an intelligent system can be constructed:

Machine knowledge representation. Conventional program restructurers hide the impact of machine knowledge on the decisions made during program restructuring as a part of the process of selecting of the heuristics used in the system. - only heuristics that are effective for the target machine are included. This is possible because only one target machine is considered. However, when the program parallelization system is designed to handle different classes of architectures, the features of the parallel computers that affect program parallelism must be abstracted and represented in a uniform structure. Separating the machine features from the heuristic acquisition process allows the description of the heuristics to be based on the machine features as well the program features. In this way, a heuristic can be applied to any target machine that has the appropriate set of features.

Program representation. The program representation problem is to define internal data structures that can encode the program's semantic and parallelism constraints. A good program representation must preserve the exact semantic and parallelism constraints of the original program. The program representation scheme must also allow easy and efficient accesses and modifications.

Transformation techniques. Transformation techniques are the essential elements of program restructuring systems. Many transformation techniques have been studied during the past two decades by a number of pioneering researchers. Rather than going through the details of the mechanical techniques for modifying program structures, in this chapter we will emphasize the heuristics for applying the transformations and the effects of the transformations on program parallelism.

Restructuring heuristics. The optimal sequences of transformations needed to get good performance from a section of code is very dependent on the program and the target machine. There are no algorithms that provide the optimal sequence of transformations for all circumstances. Heuristics are usually used to perform the task and these heuristics are usually based on the particular application and make assumptions about the target machine. In order to make the heuristic general the special features of the program and the assumptions about the machine must to be made clear.

The representation and organization of transformation knowledge. The representation, organization, and integration of the transformation knowledge are the central issues for an automatic program parallelizing system. They actually

determine the effectiveness and efficiency of the system.

Parallelism metrics. Parallelism metrics are used to compare the effects of different transformations and to decide when to terminate the optimization process. Measuring the achievable parallelism of a program on a target machine must be based on the parallelism features that the machine provides and the matching between the program structure and the target machine.

The remainder of this chapter is organized into three sections. In section 12.2, we formally define the program parallelism optimization process and discuss the machine knowledge representation problem. The program representation problem and the problem of defining parallelism metrics are also briefly discussed. In section 12.3, the transformation knowledge representation problem and some program restructuring heuristics are presented. Examples that describe the work of the inference engine are also included. In section 12.4 we give a brief summary and describe the status of our project.

12.2 Abstracting Machine Features and Building the Knowledge Base

In this section, we define the program parallelism optimization process. A machine feature abstraction scheme is introduced and a function to estimate the matching between the program level parallelism and the machine level parallelism is also given.

12.2.1 Parallelism and Program Parallelization

Parallelism can be exploited at three different levels: the algorithm level, the program level, and the machine level. Each of these three levels has a conceptual concurrency model of computation and we call this model the *virtual machine* for that level.

At the algorithm level, the virtual machine is the computational model (e.g. mesh, hyper-cube, etc.) that the parallel algorithms are based upon. *Algorithm level parallelism* can be characterized as the number of virtual processors, the complexity of inter-processor communications, and the complexity class of the parallel execution time on the virtual machine model when expressed as a function of problem size.

At the program level, each parallel programming language defines a virtual machine by the semantics of its parallel control constructs. *Program level*

parallelism can be characterized by the control and data dependence constraints imposed by the language and the user's choice of data structures.

Machine level parallelism is the maximum concurrent execution capacities of the architecture and can be characterized by various machine features.

When mapping problems from the algorithm level to the program level or from the program level to the machine level, the differences in the computational models of the two levels may cause parallelism to be lost. For example, when an algorithm is translated into a program, the concurrent properties of the algorithm may be serialized by the dependence relations inherited from program constructs and data synchronization. In some cases, the concurrency is lost because the limited parallel constructs provided by the programming language simply can not express the full parallelism in the algorithm. The problems encountered in translating parallelism from the algorithm level to the program level fall into the scope of parallel programming language design will not be discussed in this chapter.

When the program is mapped from the program level to the machine level, the programs may have to be restructured, since some specific program structures or data structures may suit the target machine better than others. *Program restructuring* is the process of improving the match between the program level parallelism and the machine level parallelism by applying a sequence of program transformations to restructure the program.

12.2.2 Program Realization and Restructuring

The process of optimizing program parallelism consists of two steps: the program restructuring process and the program realization process. The program restructuring process improves the program parallelism by modifying the structure of the program representation. The program realization process maps the programs onto the computational model of the target machine by effectively utilizing the concurrency potential of the machine.

Program level parallelism can be divided into three concurrency levels: *task, micro task,* and *operation.* At the task level, a program is decomposed into large processes which may be run on different processors. At the operation level, vector operations or scalar operations are the units of computation. The size of the vector operation represents the degree of concurrency of this level. The micro task level is the level between task level and operation level and is often characterized by loop bodies. More specifically, inside a task, operations are grouped into micro tasks, which are the blocks of code that are executed between synchronization points.

Based on the dependence constraints of the program and the feature descriptions of the target machine, the program realization process partitions the program into operation blocks and composes them to form vector operations, micro tasks, tasks, and processes. Abstractly, the process can be viewed as a function:

$$Program_realization: Computational_model\ X\ Programs \rightarrow Program_A$$

where elements of $Program_A$ are programs that are augmented with parallelism and run-time information such as processor assignments, synchronization, vectorizable or parallelizable loops, etc.

The program realization process does not actually improve the true parallelism of the program. It simply takes the current form of the computation, as represented by the program, and based on the features of the target machine, applies a mapping to realize the program into parallel form. For example, for multiprocessor systems, the outermost parallelizable loop is always used to generate tasks. For machines with vector capability, the innermost loop is the one that is vectorized (if it is legal to do so). The synchronization technique that is provided by the computational model is used to satisfy any data dependence not already satisfied by sequential execution of parts of the program.

The program restructuring process improves the match between the program level parallelism and the machine level parallelism by modifying program structure and improving the data locality in the program. In particular, it involves techniques such as changing the instruction execution order (by forward substitutions, statement reordering, etc), modifying program control (by loop interchange, loop distribution, etc), and eliminating unnecessary data accesses and modification (by data localization, block transfer, cache optimization, dead code elimination, etc). Each individual technique used to modify the structure of the program is called a *transformation.*

Abstractly, a program transformation, T, is a mapping

$$T: Program \rightarrow Program$$

that maps a program representation to a new program representation that has the same input-output semantics. The *precondition* of a transformation is the list of conditions that must be satisfied so that the result of the transformation will have the same meaning as the original program. If a program satisfies the precondition of a transformation, we say that the transformation is *applicable* to the program.

Program transformations are just mechanical techniques for changing the structure of the program. To have a positive effect on the performance, the transformations must be chosen based on the full knowledge of the program, the target machine, and a set of effective heuristics. The program restructuring

process is a composite function of a sequence of transformations. It uses heuristics that are based both on features of the program and the machine to guide the transformations and effectively translate the program into optimal form. Abstractly, it takes the form

Program_Restructuring: *Program X Computational_model X Heuristics* → *Program*

At the heart of the program restructuring is the set of rules in the knowledge base that represents the expertise about program constructs, transformation techniques, machine parallelism, and heuristics for improving the matching between programs and machines. These rules decide the effectiveness of the program restructuring process.

12.2.3 Problems in Program Parallelism Optimization

Corresponding to the concurrency levels of the program parallelism, the task of improving program parallelism can be subdivided into the following problems:

Partitioning problem. How does one partition a problem into tasks and micro tasks and form good vector operations? If the current structure of the program does not suit the hardware, various transformation techniques should be used to improve the program structures and to achieve a better partition.

Synchronization problem. When mapping a sequential program to a multiprocessors machine, the proper synchronization operations must be inserted in the code to preserve the meaning of the original program. Synchronization costs penalize the program performance, and, in the worst case, it may serialize the whole computation. Fewer synchronization points mean less processor idling time and better system performance. Grouping closely related micro tasks into one task, copying repeatedly used data into local memories, and changing data access patterns may have a positive effect on minimizing the synchronization cost.

Scheduling problem. The scheduling of the processes is another important factor in obtaining optimal performance. Traditionally, this problem is viewed as the task of the operating system. However, studies have shown that static estimates done at compile time can simplify the task of the operating system at run time [6]. There are techniques (e.g. do-across) that can estimate the required minimum process delay time and significantly reduce the amount of time the processor in "busy-wait" loops. Run-time tests can also be generated at compile time to guide the execution of the process.